Programmierpraktikum
Sommersemester 2020
Meilenstein 1 (18.05.2020)

M.Sc. Laslo Hunhold
Dept. Math./Inf., Abt. Informatik
Universität zu Köln

Bearbeitung im Git-Repository bis zum 22.06.2020 (12:00 Uhr).

Erstellen Sie im Git-Repository den Ordner `ms1`, in dem Sie Ihre Lösung zu diesem Meilenstein erstellen werden. Jede in diesem Ordner erzeugte Klasse wird damit dem Package `ms1` zugewiesen. Dies müssen Sie auch explizit mit einer `package`-Anweisung angeben, da es sonst beim Kompilieren zu einer Fehlermeldung kommt.

Einführung

Die Motivation für diesen Meilenstein ist die Entwicklung eines Programms, welches auch im Alltag nützlich ist und einen Mehrwert bringt. Ein Beispiel für ein solches Programm ist ein Taschenrechner, der bei kleineren gelegentlichen Berechnungen hilfreich ist.
Die heutzutage häufigste Verwendung von Taschenrechnern erfolgt auf mobilen Endgeräten. Die Funktionen reichen aus, um beispielsweise Beträge zu addieren oder Prozentsätze auszurechnen, aber stoßen an ihre Grenzen, wenn es um die Lösung komplexerer Ausdrücke geht, auf die Sie in Ihrer Studienlaufbahn sicherlich auch schon gestoßen sind oder stoßen werden. Zu diesem Zweck werden dann häufig der wissenschaftliche Taschenrechner, ein CAS, Internetdienste oder anderes verwendet, was aber jeweils mit gewissen Einschränkungen verbunden ist. Vor allen Dingen handelt es sich bei diesen Einschränkungen um welche in Bezug auf Benutzbarkeit („Wie schnell kann ich meine Gleichung in den Rechner übertragen?“) und Genauigkeit („Inwieweit kann ich dem Ergebnis trauen?“).
Gegenstand dieses Meilensteins ist die Entwicklung eines Taschenrechners, der sowohl eine einfache und schnelle Benutzbarkeit als auch eine zweifelsfreie beliebige Genauigkeit als Zielsetzung hat.

Postfixnotation

In Bezug auf die Benutzbarkeit ist eine Leitfrage und ein häufig ignorierter Aspekt, wie sich ein gegebener Ausdruck möglichst schnell in den Rechner übertragen läßt. Betrachten wir zum Beispiel den folgenden Ausdruck:

$$(2 + 4) * (6 + 9) \tag{1}$$

Es ist auffällig, daß wir relativ viele Klammern setzen müssen, um die Beziehungen zwischen den Operatoren und Operanden korrekt wiederzugeben. Mehr Klammern bedeuten natürlich auch mehr notwendige Eingaben. Die Ursache für die Notwendigkeit von Klammern sind Präzedenzregeln von Operatoren, z.B. Punkt-vor-Strich, welche häufig mit Klammern „überschrieben“ werden müssen. Aber sind Klammern und Präzedenzregeln überhaupt notwendig?
In den 1920er Jahren entwickelte der polnische Mathematiker Jan Łukasiewicz eine alternative Schreibweise von Ausdrücken. Anstatt zum Beispiel beim Ausdruck „1 + 2“ den Operator „+“ zwischen seine Operanden „1“ und „2“ zu schreiben („Infixnotation“), wird der Operator vor seine Operanden geschrieben und wir erhalten „+ 1 2“. Diese alternative Schreibweise bezeichnet man als „Präfixnotation“, da der Operator ein Präfix der Operanden ist. Später entwickelte sich eine Ableitung, die „Postfixnotation“, bei der der Operator hinter seine Operanden geschrieben wird, in diesem Fall also „1 2 +“.
Wandeln wir Ausdruck (1) in Postfixnotation um, erhalten wir:

$$2\ 4\ +\ 6\ 9\ +\ * \tag{2}$$

Der enorme Vorteil dieser Notation ist, daß es keine Präzedenzregeln zwischen Operatoren gibt. Die Auswertungsreihenfolge wird allein durch die Notation an sich vorgegeben. Die Notwendigkeit von Klammern entfällt damit und wir brauchen nur 7 statt 11 Eingaben, eine Ersparnis von immerhin knapp 40%. Dies sehen wir noch deutlicher an dem folgenden Beispiel. Den Ausdruck

$$\ln\left(\frac{1+\sqrt{1+\sin(5)}}{2+\exp(\pi)}\right) \tag{3}$$

würde man in Infixnotation mit 23 Eingaben als

$$\ln\ (\ (\ 1\ +\ \mathrm{sqrt}\ (\ 1\ +\ \sin\ (\ 5\ )\ )\ )\ /\ (\ 2\ +\ \exp\ (\ \pi\ )\ ) \tag{4}$$

ausdrücken. Derselbe Ausdruck in Postfixnotation ist

$$1\ 5\ \sin\ +\ \mathrm{sqrt}\ 1\ +\ 2\ \pi\ \exp\ +\ /\ \ln \tag{5}$$

und ergibt sich schon mit 13 Eingaben, in diesem Fall eine Ersparnis von knapp 45%.
Ein Nachteil der Postfixnotation ist aber vor allen Dingen die notwendige Umgewöhnung, welche besonders zu Beginn die Lesbarkeit beeinträchtigt. Die Infixnotation, Präzedenzregeln und Klammerung werden schon in der Grundschule vertieft und sind im Alltag etabliert. Trotz dessen ist die Wahl der Postfixnotation insofern kanonisch für unseren Anwendungsfall, da sie die Eingabekomplexität massiv verringert.

Reihendarstellungen und Auswertungen von Funktionen

Zum Zwecke der Genauigkeit werden wir die in Meilenstein 0 eingeführte Langzahlarithmetik zur Berechnung verwenden. Bei der Arbeit mit `java.math.BigDecimal` ist Ihnen aber sicherlich schon schnell aufgefallen, daß keine elementaren Funktionen (ln, exp, sin, cos, tan, etc.) bereitgestellt werden. Stattdessen stehen uns unter anderem nur die Grundrechenarten (Addition, Subtraktion, Multiplikation und Division), eine Quadratwurzeloperation und eine Ganzzahlpotenzierung zur Verfügung.
Es ist möglich, allein mit Grundrechenarten jede dieser elementaren Funktionen über ihre Reihenentwicklungen zu berechnen. Die Herangehensweise für jede zu implementierende Funktion wird im Abschnitt „Operatoren“ näher beleuchtet, jedoch werden wir hier schon einmal die Exponentialfunktion als Beispiel betrachten und festhalten, welche Dinge bei der Berechnung zu beachten sind.
Eine Reihenentwicklung ist die Darstellung einer beliebig oft differenzierbaren Funktion in der Variablen x (z.B. $\exp(x)$) als Reihe (d.h. als eine unendliche Summe). Die Summanden sind üblicherweise skalierte (also mit einem Vorfaktor versehene) Potenzen der Variablen (also $1, x^2, x^3, \dots$), können aber auch andere Ausdrücke in Abhängigkeit von x sein. Die bekannteste Form der Reihenentwicklung ist die TAYLORreihe, welche für die Exponentialfunktion (hier im Entwicklungspunkt 0)

$$\exp(x) := \sum_{k=0}^{\infty} \frac{1}{k!}x^k = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \dots \tag{6}$$

lautet. Wir sehen, daß die Summanden dieser Reihe mit $1/k!$ skalierte Potenzen x^k von x sind, und wir können in diesem Fall beliebige Werte von x in die Summe einsetzen, um $\exp(x)$ zu bestimmen. Die Auswertung der Summe an sich benötigt nur Grundrechenarten (Addition, Division, Multiplikation). Theoretisch haben wir also das Problem gelöst und einen Weg gefunden, nur mit Grundrechenarten die Exponentialfunktion auszuwerten. Übertragen auf eine reale Berechnung verbleiben aber zwei Probleme:
Das erste Problem ist, daß es sich hierbei um eine unendliche Summe handelt, und wir nicht unendlich viele Operationen in endlicher Zeit durchführen können. Es bleibt uns nichts anderes

übrig, als die Summe an einer bestimmten Stelle abzubrechen und damit nur die $N \geq 0$ ersten Summanden zu addieren, also

$$\exp(x) \approx \sum_{k=0}^{N} \frac{1}{k!} x^k = 1 + x + \frac{1}{2}x^2 + \cdots + \frac{1}{N!}x^N. \tag{7}$$

Natürlich weicht diese abgebrochene Summe vom korrekten Wert ab, und es ist möglich, mit einer sogenannten „Restgliedabschätzung“ diesen Fehler abzuschätzen. Das Restglied ist nichts anderes als die Summe der übrigen Faktoren, die man bei der abgebrochenen Summe weggelassen hat, also

$$\sum_{k=N+1}^{\infty} \frac{1}{k!} x^k. \tag{8}$$

Man möchte sicherstellen, daß der Abbruch der Summation beim Index N nicht das Ergebnis über ein bestimmtes Maß $\varepsilon > 0$ (die „Fehlertoleranz“) hinaus beeinflußt. Wenn man beispielsweise eine Fehlertoleranz von $\varepsilon = 10^{-3}$ hat, muß

$$\left| \sum_{k=N+1}^{\infty} \frac{1}{k!} x^k \right| < 10^{-3} \tag{9}$$

gelten. Der Wert des Restglieds ist natürlich nicht genau bestimmbar, da dies voraussetzen würde, die unendliche Summe auszuwerten, aber es läßt sich mit dem Satz von Taylor eine obere Schranke für den Betrag des Restglieds finden. Wenn diese obere Schranke unter die Fehlertoleranz fällt, weiß man dann auch, daß das Restglied diesen Wert unterschreitet. Für $x \in (-1, 1)$ folgt aus dem Satz von Taylor (Herleitung weggelassen), daß

$$\left| \sum_{k=N+1}^{\infty} \frac{1}{k!} x^k \right| < \frac{e}{(N+1)!} \tag{10}$$

gilt. Dies bedeutet, daß sobald $e/(N+1)!$ bei einem bestimmten N unsere Fehlertoleranz ε unterschreitet, das Restgliedb betraglich diese nicht überschreitet. Mit dem Ansatz, die Summierung an einem bestimmten Punkt abzubrechen, während wir gleichzeitig sicherstellen, daß dieser Abbruch unser Ergebnis im Rahmen unserer Genauigkeit nicht beeinträchtigt, lösen wir also das erste Problem.
Das zweite Problem bei der Reihenauswertung ist die Tatsache, daß die Werte von x beliebig groß sein können (da der Anwender beliebige Werte einsetzen kann). Dies hat zur Folge, daß auch die Summanden der Reihenentwicklung nur langsam kleiner werden und womöglich erst für ein sehr großes N unsere Fehlertoleranz ε unterschreiten können. Zwar haben wir eine gewisse Toleranz, was die Auswertungsgeschwindigkeit angeht, da wir bei einem einzelnen Rechenausdruck nur relativ wenige Operatoren auswerten müssen, aber extrem große Werte für x können einen spürbaren Unterschied machen.
Um dieses Problem zu umgehen, versucht man den Wertebereich des in die Reihenentwicklung eingesetzten x zu beschränken (z.B. auf das Intervall $[-1, 1]$). Dies erreicht man damit, daß man Eigenschaften der verschiedenen elementaren Funktionen ausnutzt. Bei der Exponentialfunktion nutzt man zum Beispiel die Tatsache aus, daß es zu jedem $x \in \mathbb{R}$ ein $a \in (-1, 1)$ und $b \in \mathbb{N}$ gibt, sodaß $x = a \cdot 10^b$. Es folgt hiermit

$$\exp(x) = \exp(a \cdot 10^b) = \exp(a)^{10^b}. \tag{11}$$

Egal wie groß x ist, setzt man in die Reihe nur den Wert $a \in (-1, 1)$ ein und potenziert das Ergebnis ganzzahlig mit 10^b. Auf diese Art lösen wir das Problem großer x-Werte und erhalten eine stabile Methode zur Auswertung der Exponentialfunktion. Darüber hinaus, da wir die

input : Auswertungsstelle $x \in \mathbb{R}$
Relative Fehlertoleranz $\varepsilon > 0$
output: Näherungswert von $\exp(x)$

$b \leftarrow$ kleinstes k sodaß $x < 10^k$;
$a \leftarrow x/10^b$;
$r \leftarrow 0$;
$k \leftarrow 0$;
while $e/(k+1)! \geq \varepsilon$ **do**
  $r \leftarrow r + a^k/k!$;
  $k \leftarrow k + 1$;
end
while $b > 0$ **do**
  $r \leftarrow r^{10}$;
  $b \leftarrow b - 1$;
end
return r;

Algorithmus 1: Verfahren zur näherungsweisen Auswertung von $\exp(x)$.

Exponentialfunktion nur an Stellen aus $(-1, 1)$ auswerten, können wir die Restgliedabschätzung aus (10) verwenden. Das gesamte Verfahren zur Auswertung der Exponentialfunktion finden Sie in Algorithmus 1 als Pseudocode. Es muß hier angemerkt werden, daß die Fehlertoleranz in Algorithmus 1 und folgenden Algorithmen nicht direkt eingefordert werden kann. Dies liegt daran, daß diese Toleranz nur bei der Auswertung des skalierten Eingabewerts eingefordert wird, und die ausgewertete Reihe danach noch skaliert wird (mit der Potenzierung um 10 und b). Aus diesem Grund wird bei den Algorithmen unter den Eingabedaten von einer „relativen Fehlertoleranz“ gesprochen.
Dies ist aber kein Nachteil hier, da die Skalierungen nur relativ wenige Operationen darstellen und nur geringe Rundungsfehler entstehen. Die Fehlertoleranz ist in der Mathcore-Klasse auf Seite 12 so gegeben, daß die Auswertung der Reihe auf alle internen Stellen exakt ist. Kurzum gibt es trotz dieser Skalierungsfrage keine Probleme mit dem Ergebnis.

Überblick

Nach der Einführung in die Postfixnotation und die Herangehensweise der Auswertung von Funktionen werfen wir nun einen Blick auf die Struktur des Programms. Der Rechner sollte aus drei Klassen bestehen, die in den folgenden Abschnitten genau spezifiziert werden. Die erste Klasse ist die GUI-Klasse für Funktionen in Bezug auf die graphische Benutzeroberfläche. In der zweiten Klasse, Postfix, findet die Implementierung des Postfix-Parsers statt. Die Mathcore-Klasse dient der Implementierung des numerischen Unterbaus, also Operatoren, Konstanten und Hilfsfunktionen.

GUI

Ein Taschenrechner sollte drei Grundfunktionen haben: Ein Ein- und Ausgabefeld, einen Verlauf vorheriger Berechnungen und die Möglichkeit der Ausgabe von Fehlermeldungen. Die meisten dieser Funktionen lassen sich nicht mit einer reinen Darstellung bereitstellen, sondern müssen auch durch Interaktionen des Benutzers mit der Benutzeroberfläche realisiert werden. Die Entwicklung eines solchen interaktiven GUIs in Java erfolgt über sogenannte Listener-Funktionen und erschwert die schon ohnehin komplexe Arbeit mit den Java GUI-Bibliotheken.
Da sich die GUI-Entwicklung bei jeder Programmiersprache bzw. Toolkit häufig grundlegend un-

terscheidet und der Lerneffekt gering ist, wird im Interesse des Fokus auf wesentlichere Aspekte eine GUI-Klasse auf Seite 11 bereitgestellt, die Sie für Ihre Abgabe frei verwenden und modifizieren dürfen.
Die Anbindung an den Postfix-Parser erfolgt bei der GUI-Klasse in Zeile 44. Wie später noch beschrieben wird, nimmt die `eval()`-Funktion ein `String`-Argument (den Rechenausdruck) an gibt die Lösung als `String` zurück. Bei einer fehlerhaften Eingabe oder anderen Fehlern wirft die Funktion eine `Exception`, die in der GUI-Klasse abgefangen und automatisch in der Benutzeroberfläche angezeigt wird.
Sinnvolle Verbesserungen und Erweiterungen dieses Grundgerüsts werden begrüßt und können sich positiv auf die Endbewertung auswirken, falls es andere Defizite gibt. Solche Änderungen sind aber nicht erforderlich, um die Bestnote zu erreichen. Versuchen Sie also nicht, die GUI grundlos zu modifizieren, sondern führen Sie dies nur durch, falls Sie hierzu eine sinnvolle Idee haben und es sich zutrauen.

Postfix

Wie bereits eingeführt hat die Postfix-Klasse die Aufgabe, einen Postfix-Rechenausdruck (als `String`) auszuwerten. Dies erfolgt über die Bereitstellung einer einzelnen Funktion, `eval()`, die einen `String` als Argument annimmt und einen `String` zurückgibt.
Ein Postfix-Rechenausdruck hat eine bestimmte Form. Hierzu betrachten wir hierzu die Ausdrücke (2) und (5). Wir sehen, daß die einzelnen Operatoren und Operanden durch (ein oder mehrere) Leerzeichen getrennt sind. Insofern kann man die Operatoren und Operanden als Token interpretieren, die von links nach rechts eingelesen werden. Bei jedem Token wird zuerst geprüft, ob es sich um einen Operator oder einen Operanden (also eine Zahl) handelt. Im Falle einer Zahl wird diese als `BigDecimal` eingelesen und auf einen `BigDecimal`-Stapelspeicher gelegt, den wir als „Operandenstapel“ bezeichnen. Wird statt einer Zahl ein Operator angetroffen, werden so viele Elemente vom Operandenstapel genommen, wie der Operator benötigt, und an die Operatorfunktion übergeben, die sich in der Mathcore- Klasse befindet. Das Ergebnis der Operation wird wieder auf den Operandenstapel gelegt, oder eine Exception weitergegeben, die in der Operatorfunktion geworfen wurde.
Wurden alle Tokens verarbeitet wird der Operandenstapel geprüft. Enthält dieser kein oder mehr als ein Element, war der Ausdruck fehlerhaft und es sollte eine Exception geworfen werden. Enthält der Operandenstapel nur ein Element, so ist dieses das Endergebnis der Berechnung. Dies wird auf 30 Stellen gerundet (siehe `mc_out` in der Mathcore-Klasse), in einen `String` umgewandelt und zurückgegeben.
Einen Spezialfall stellen die Token „`pi`“ bzw. „`e`“ als Konstanten π (Kreiszahl) bzw. e (EULERsche Zahl) dar, die der Parser bei einer Begegnung mit dem jeweiligen Wert auf den Operandenstapel legen soll. Beide Zahlen auf 2000 Stellen genau finden Sie im Grundgerüst der Mathcore-Klasse auf Seite 12.
Liegen nicht genug Elemente auf dem Stack oder ist ein Token weder als Operator noch als Zahl identifizierbar, sollte natürlich auch eine Exception geworfen werden, die den Fehler beschreibt. Ihnen steht es frei, Ihre eigenen Stack- (nach entsprechender Modifikation) und Tokenizer-Implementierungen aus Meilenstein 0 (durch `import ms0.a0.Stack` bzw. `import ms0.a1.Tokenizer`) oder entsprechende Werkzeuge aus der Standardbibliothek zu verwenden. Fehlerhafte Implementierungen in Meilenstein 0 dürfen Sie natürlich auch korrigieren, was aber nicht als Leistung im Meilenstein 1 gewertet wird.

Mathcore

In der Mathcore-Klasse sollen die numerischen Aspekte des Rechners implementiert werden. Sie finden auf Seite 12 eine Grundgerüst dieser Klasse, die schon ein paar nützliche Konstanten bereitstellt. Dies dürfen Sie wie die GUI-Klasse frei verwenden und modifizieren.

Alle internen Berechnungen sollen mit 200 Stellen arbeiten (siehe MathContext `mc` in Mathcore.java). Erst bei der finalen Ausgabe am Ende der `eval()`-Funktion soll auf 30 Stellen gerundet werden (siehe MathContext `mc_out` in Mathcore.java). Der Grund für diese große Diskrepanz ist die Tatsache, daß trotz größter Sorgfalt natürlich jede einzelne Rechenoperation Rundungsfehler einführt. Die Einbuße kann selbst bei langen Ausdrücken lediglich 1 bis 3 Stellen an Genauigkeit bedeuten (und damit schon eine interne Stellenzahl von 40 ausreichend sein lassen), aber das Ziel für dieses Projekt ist eine zweifelsfreie Genauigkeit. Die sehr hohe Präzision der internen Berechnungen bringt aufgrund der geringen Zahl von Operationen auch keinen spürbaren Nachteil, und erspart uns alternative Herangehensweisen wie komplexe Fehleranalysen.
Wie im vorherigen Abschnitt beschrieben ruft die `eval()`-Funktion für jeden angetroffenen Operator seine zugehörige Operatorfunktion auf. Die Operatorfunktionen werden in dieser Mathcore-Klasse implementiert. Die genaue Form (Anzahl der Operanden, Exception-Würfe) der Operatorfunktionen wird in den folgenden Abschnitten für jede zu implementierende Operatorfunktion genauer beschrieben. In dieser Auflistung finden Sie jeweils den Identifikator des zu implementierenden Operators (z.B. „`+`“ bei Addition) und eine verkürzte Methodensignatur (hier z.B. „`add(a,b)`“, die auch die Anzahl der Operanden verrät (in diesem Fall 2). Die Typangaben der Parameter und des Rückgabewerts fehlen hier, da jede Operatormethode nur `BigDecimal`s annimmt und einen `BigDecimal` zurückgibt. In diesem Beispiel wäre damit der korrekte Methodenkopf „`public BigDecimal add(BigDecimal a, BigDecimal b) throws Exception`“.

Elementare Rechenoperationen

- *Addition* (`+`, `add(a,b)`): Berechne die Summe von `a` und `b`.
- *Subtraktion* (`-`, `sub(a,b)`): Ziehe `b` von `a` ab.
- *Division* (`/`, `div(a,b)`): Teile `a` durch `b` und werfe eine Exception, falls $b = 0$.
- *Multiplikation* (`*`, `mul(a,b)`): Berechne das Produkt von `a` und `b`.

Fakultät

Der Fakultätsoperator ist einer der wenigen Operatoren, die auch sonst immer in Postfix-Notation geschrieben werden. Insofern ist die Benutzung dieses Operators im Rechner intuitiv.

- *Fakultät* (`!`, `fak(a)`): Berechne die Fakultät von a, falls a ganzzahlig ist, und werfe sonst eine Exception.

Exponentialfunktion

- *Exponentialfunktion* (`exp`, `exp(a)`): Werte die Exponentialfunktion an der Stelle a aus.

Hierzu gab es bereits eine Herleitung in der Einführung und eine Angabe als Pseudocode in Algorithmus 1.

Logarithmen

Der natürliche Logarithmus kann genau wie die Exponentialfunktion über eine Reihendarstellung ausgewertet werden. Der Hauptunterschied zur Exponentialfunktion ist, daß der Logarithmus nur für streng positive Zahlen auswertbar ist. Die folgende Reihe läßt sich aus einem Zusammenhang des natürlichen Logarithmus mit dem Areatangens hyperbolicus herleiten, was hier aber aus Übersichtsgründen weggelassen wird. Es gilt

$$\log(x) := \sum_{k=0}^{\infty} \frac{2}{2k+1} \cdot \left(\frac{x-1}{x+1}\right)^{2k+1}, \tag{12}$$

und wir können schon abschätzen, daß die Summanden besonders klein werden, falls x nahe 1 ist. Um dies zu erreichen, wenden wir auch hier wieder einen Trick an. Es gilt nämlich

$$\ln(x) = 2 \cdot \frac{1}{2} \cdot \ln(x) = 2 \cdot \ln(\sqrt{x}), \tag{13}$$

und da, falls die Quadratwurzel mehrfach auf eine beliebige streng positive Zahl angewendet wird, diese immer näher an die 1 gebracht wird, müssen wir einfach so oft die Wurzel aus unserem Eingabewert x ziehen, bis sich das Ergebnis a in einem engen Intervall um 1 (z.B. $[0.9, 1.1]$) befindet. Danach werten wir die Reihe an der Stelle a aus und multiplizieren das Endergebnis so oft mit 2, wie wir zuvor die Quadratwurzel angewendet haben. Damit erhalten wir dann den natürlichen Logarithmus von x. Wird x auf $[0.9, 1.1]$ eingeschränkt, so ergibt sich auch die Restgliedabschätzung nach dem Satz von Taylor (Herleitung weggelassen) als

$$\left| \sum_{k=N+1}^{\infty} \frac{2}{2k+1} \cdot \left(\frac{a-1}{a+1} \right)^{2k+1} \right| \leq \frac{1}{180 \cdot (2N+3) \cdot 19^{2N+1}}. \tag{14}$$

Das gesamte Vorgehen wird in Algorithmus 2 zusammengefaßt als Pseudocode beschrieben.

input : Auswertungsstelle $x > 0$
Relative Fehlertoleranz $\varepsilon > 0$
output: Näherungswert von $\ln(x)$

$a \leftarrow x$;
$w \leftarrow 0$;
while $|1 - a| \geq 0.1$ **do**
  $a \leftarrow \sqrt{a}$;
  $w \leftarrow w + 1$;
end
$r \leftarrow 0$;
$k \leftarrow 0$;
while $1/[180 \cdot (2k+3) \cdot 19^{2k+1}] \geq \varepsilon$ **do**
  $r \leftarrow r + \frac{2}{2k+1} \cdot \left(\frac{a-1}{a+1} \right)^{2k+1}$;
  $k \leftarrow k + 1$;
end
while $w > 0$ **do**
  $r \leftarrow 2r$;
  $w \leftarrow w - 1$;
end
return r;

Algorithmus 2: Verfahren zur näherungsweisen Auswertung von $\ln(x)$.

Bisher haben wir nun aber erst nur den natürlichen Logarithmus berechnet. Die Logarithmen zu anderen Basen lassen sich aber leicht daraus bestimmen. Wir nutzen hierfür die Basiswechselgleichung

$$\log_m(x) = \frac{\log_n(x)}{\log_n(m)}. \tag{15}$$

Es folgt also, daß wir jeden beliebigen Logarithmus von x zu einer Basis b berechnen können, indem wir den natürlichen Logarithmus von x durch den natürlichen Logarithmus von b dividieren. Die folgenden drei Logarithmen sollen im Rechner verfügbar sein:

- *Natürlicher Logarithmus* (`ln`, `ln(a)`): Werte den natürlichen Logarithmus (Basis e) an der Stelle a aus.

- *Dekadischer Logarithmus* (`lg`, `ln(a)`): Werte den dekadischen Logarithmus (Basis 10) an der Stelle a aus.
- *Allgemeiner Logarithmus* (`log`, `ln(a,b)`): Werte den Logarithmus zur Basis a an der Stelle b aus. Werfe eine Exception, falls a kleiner gleich 0 oder gleich 1 ist.

Natürlich sollten alle Logarithmus-Funktionen eine Exception werfen, falls das Argument kleiner gleich 0 ist.

Potenzierung

Für die Potenzierung a^b betrachten wir nur den Fall, daß a positiv ist. Hier nutzen wir die Tatsache aus, daß 0^b immer 0 ist (für $b > 0$) und für $a > 0$ die Gleichung

$$a^b = \exp(\ln(a^b)) = \exp(b \cdot \ln(a)) \tag{16}$$

gilt, wir also die Potenz leicht mit der Exponentialfunktion und dem Logarithmus berechnen können. Die Implementierung sollte wie folgt aussehen:

- *Potenz* (`^`, `pot(a,b)`): Berechne die Potenz a^b, und werfe eine Exception, falls a negativ ist oder $a = 0$ und $b \leq 0$.

Man kann auch negative Zahlen allgemein potenzieren, wenn der Exponent eine rationale Zahl mit ungeradem Nenner ist, aber dies wird hier nicht näher betrachtet.

Wurzel

Die `BigDecimal`-Klasse stellt bereits eine Quadratwurzelfunktion (`sqrt()`) zur Verfügung. Die Berechnung allgemeiner Wurzeln kann jedoch auch bewerkstelligt werden, indem man die zuvor implementierte Potenzierungsfunktion verwendet. Allgemein gilt nämlich für $p > 0$

$$\sqrt[p]{a} = a^{1/p}, \tag{17}$$

sofern $a \geq 0$. Die beiden Wurzeloperatoren sollten wie folgt lauten:

- *Quadratwurzel* (`sqrt`, `sqrt(a)`): Berechne die Quadratwurzel von a und werfe eine Exception, falls $a < 0$.
- *Wurzel* (`root`, `root(a,b)`): Berechne die `a`-te Wurzel von b und werfe eine Exception, falls $a < 0$ oder $p \leq 0$.

Trigonometrische Funktionen

Das Vorgehen hier ist wie bei der Exponentialfunktion und beim Logarithmus. Wir betrachten die Reihenentwicklung

$$\sin(x) := \sum_{k=0}^{\infty} (-1)^k \cdot \frac{x^{2k+1}}{(2k+1)!} \tag{18}$$

und versuchen, die Eingabewerte nahe bei 0 halten. Wir nutzen hier aus, daß der Sinus 2π-periodisch ist (d.h. es gilt $\sin(x) = \sin(x + 2\pi)$ für alle $x \in \mathbb{R}$) und berechnen den (positiven) Rest a einer Division von x durch 2π. Dieser Rest a liegt nun also in $[0, 2\pi)$. Um noch näher an die Null heranzukommen, subtrahieren wir 2π von a, falls a in $(\pi, 2\pi)$ liegt (was am Ergebnis nichts ändert, da Sinus eben 2π-periodisch ist). Letztendlich liegt unser a also in $(-\pi, \pi]$, und es gilt $\sin(a) = \sin(x)$. Mit der Einschränkung auf $(-\pi, \pi)$ ergibt sich mit dem Satz von Taylor (Herleitung weggelassen) die Restgliedabschätzung

$$\left| \sum_{k=N+1}^{\infty} (-1)^k \cdot \frac{a^{2k+1}}{(2k+1)!} \right| < \frac{\pi^{2N+3}}{(2N+3)!}. \tag{19}$$

input : Auswertungsstelle $x \in \mathbb{R}$
Relative Fehlertoleranz $\varepsilon > 0$
output: Näherungswert von $\sin(x)$

$a \leftarrow x \mod 2\pi$;
if $a \in (\pi, 2\pi)$ **then**
  $a \leftarrow a - 2\pi$;
end
$r \leftarrow 0$;
$k \leftarrow 0$;
while $\pi^{2k+3}/(2k+3)! \geq \varepsilon$ **do**
  $r \leftarrow r + (-1)^k \cdot \frac{a^{2k+1}}{(2k+1)!}$;
  $k \leftarrow k + 1$;
end
return r;

Algorithmus 3: Verfahren zur näherungsweisen Auswertung von $\sin(x)$.

Das gesamte Verfahren ist in Algorithmus 3 zusammengefaßt.
Andere trigonometrische Funktionen lassen sich direkt vom Sinus ableiten. Der Kosinus ist lediglich eine horizontale Verschiebung und Spiegelung des Sinus, nämlich

$$\cos(x) = \sin(\pi/2 - x), \tag{20}$$

was die Implementierung trivial macht. Ähnlich sieht es beim Tangens aus, der als

$$\tan(x) = \frac{\sin(x)}{\cos(x)}, \tag{21}$$

definiert ist. Es muß also bei der Auswertung lediglich geprüft werden, ob $\cos(x) = 0$ ist. Zusammengefaßt ergibt sich also die Gesamtliste der zu implementierenden trigonometrischen Funktionen als

- *Sinus* (`sin`, `sin(a)`): Werte den Sinus an der Stelle a aus.
- *Kosinus* (`cos`, `cos(a)`): Werte den Kosinus an der Stelle a aus.
- *Tangens* (`tan`, `tan(a)`): Werte den Tangens an der Stelle a aus. Werfe eine Exception, falls $\cos(a) = 0$ ist.

Anforderungen

Die folgenden Anforderungen werden unter anderem an Ihre Abgaben gestellt:

- Einhaltung der vorgegebenen Verzeichnis- und Klassenstruktur (Ergänzungen von Klassen, Variablen und Methoden sind zulässig) und Implementierung aller vorgegebenen Funktionen.
- Ihr Repository enthält nur `java`-Dateien und (falls vorhanden) Dokumentation. Insbesondere ist es frei von `class`-Dateien, Eclipse- (`.metadata`, `.classpath`, `.project`, etc.) und System-Metadaten (`.DS_Store`, `Thumbs.db`, etc.)
- Die eingereichten `java`-Dateien sind im Zeichensatz UTF-8 enkodiert und kompilierbar.
- Der Quelltext ist an komplexen Stellen sinnvoll kommentiert und allgemein strukturiert.

Testausdrücke

Ein guter Weg, die eigene Implementierung zu testen, ist über Testausdrücke mit bekanntem Ergebnis. Im Folgenden ist eine Auswahl solcher gegeben.

- pi 2.424 + 430 + e −

  432.847310825130748003102355912

- pi e /

  1.155727349790921717910093183 31

- 3 ! ! !

  2.601218943565795100204903227 08E+1746

- 3 exp exp

  528491311.485494206009318412514

- 1E100 ln ln ln

  1.693632474984233054857995109 11

- 2 1000 ^ 1000 ^

  9.900656229295898250697923616 30E+301029

- 1000 1000 2 1000 ^ 1000 ^ root root

  2.000000000000000000000000000000

- pi e root

  1.374802227439358631782821879 21

- e sin

  0.410781290502908695476009492018

- e e ^ e ^ tan cos sin

  0.822303699242162245376297299985

Alle Ergebnisse sind bis auf die letzte Nachkommastelle, die gerundet ist, exakt. Testen Sie bei Gelegenheit auch einmal die Ausdrücke (natürlich umnotiert) in anderen Rechnern.

```java
package ms1;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class GUI implements ActionListener,ListSelectionListener,FocusListener {
	DefaultListModel<String> verlauf;
	JTextField input;
	JList<String> liste;
	JLabel status;

	public GUI() {
		JFrame frame = new JFrame("Rechner");
		Font fontmain = new Font("Serif", Font.BOLD, 15);
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setSize(600, 400);
		frame.setFont(fontmain);

		Font fontlarge = new Font("Serif", Font.BOLD, 30);
		input = new JTextField();
		input.setFont(fontlarge);
		input.addActionListener(this);

		verlauf = new DefaultListModel<>();
		liste = new JList<>(verlauf);
		liste.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
		liste.addListSelectionListener(this);
		liste.addFocusListener(this);
		JScrollPane sp = new JScrollPane();
		sp.setViewportView(liste);

		status = new JLabel("");

		frame.getContentPane().add(BorderLayout.NORTH, input);
		frame.getContentPane().add(BorderLayout.CENTER, sp);
		frame.getContentPane().add(BorderLayout.SOUTH, status);
		frame.setVisible(true);
	}

	public static void main(String[] args) {
		new GUI();
	}

	public void actionPerformed(ActionEvent e) {
		String result = null;
		try {
			result = Postfix.eval(input.getText());
		} catch (Exception ex) {
			status.setText(ex.getMessage());
		}
		if (result != null) {
			verlauf.insertElementAt(input.getText(), 0);
			input.setText(result);
			input.setCaretPosition(0);
			status.setText("");
		}
	}
	public void valueChanged(ListSelectionEvent e) {
		if (!e.getValueIsAdjusting() && liste.getSelectedValue() != null) {
			input.setText(liste.getSelectedValue().toString());
		}
	}
	public void focusGained(FocusEvent e) {
		return;
	}
	public void focusLost(FocusEvent e) {
		liste.clearSelection();
	}
}
```

GUI.java

```
1  package ms1;
2  import java.lang.Math;
3  import java.math.BigDecimal;
4  import java.math.MathContext;
5  import java.math.RoundingMode;
6
7  public class Mathcore {
8        public static final int STELLEN_INTERN = 200;
9        public static final int STELLEN_OUTPUT = 30;
10       public static final MathContext mc = new MathContext(STELLEN_INTERN,
11                                                            RoundingMode.HALF_EVEN);
12       public static final MathContext mc_out = new MathContext(STELLEN_OUTPUT,
13                                                                RoundingMode.HALF_EVEN);
14       public static final BigDecimal epsilon = BigDecimal.ONE.scaleByPowerOfTen(-(STELLEN_INTERN + 1));
15       public static final String PI =
16             "3.14159265358979323846264338327950288419716939937510582097494459230781640628620 8" +
17             "99862803482534211706798214808651328230664709384460955058223172535940812848111745" +
18             "02841027019385211055596446229489549303819644288109756659334461284756482337867831" +
19             "65271201909145648566923460348610454326648213393607260249141273724587006606315588" +
20             "17488152092096282925409171536436789259036001133053054882046652138414695194151160" +
21             "94330572703657595919530921861173819326117931051185480744623799627495673518857527" +
22             "24891227938183011949129833673362440656643086021394946395224737190702179860943702" +
23             "77053921717629317675238467481846766940513200056812714526356082778577134275778960" +
24             "91736371787214684409012249534301465495853710507922796892589235420199561121290219" +
25             "60864034418159813629774771309960518707211349999998372978049951059731732816096318" +
26             "59502445945534690830264252230825334468503526193118817101000313783875288658753320" +
27             "83814206171766914730359825349042875546873115956286388235378759375195778185778 05" +
28             "32171226806613001927876611195909216420198938095257201065485863278865936153381827" +
29             "96823030195203530185296899577362259941389124972177528347913151557485724245415069" +
30             "59508295331168617278558890750983817546374649393192550604009277016711390098488240" +
31             "12858361603563707660104710181942955596198946767837449448255379774726847104047534" +
32             "64620804668425906949129331367702898915210475216205696602405803815019351125338243" +
33             "00355876402474964732639141992726042699227967823547816360093417216412199245863150" +
34             "30286182974555706749838505494588586926995690927210797509302955321165344987202755" +
35             "96023648066549911988183479775356636980742654252786255181841757467289097777279380" +
36             "00816470600161452491921732172147723501414419735685481613611573525521334757418494" +
37             "68438523323907394143334547762416862518983569485562099219222184272550254256887671" +
38             "79049460165346680498862723279178608578438382796797668145410095388378636095068006" +
39             "42251252051173929848960841284886269456042419652850222106611863067442786220391949" +
40             "45047123713786960956364371917287467764657573962413890865832645995813390478027 5901";
41       public static final String E =
42             "2.71828182845904523536028747135266249775724709369995957496696762772407663035354 7" +
43             "59457138217852516642742746639193200305992181741359662904357290033429526059563073" +
44             "81323286279434907632338298807531952510190115738341879307021540891499348841675092" +
45             "44761460668082264800168477411853742345442437107539077744992069551702761838606261" +
46             "33138458300075204493382656029760673711320070932870912744374704723069697720931014" +
47             "16928368190255151086574637721112523897844250569536967707854499699679468644549059" +
48             "87931636889230098793127736178215424999229576351482208269895193668033182528869398" +
49             "49646510582093923982948879332036250944311730123819706841614039701983767932068328" +
50             "23764648042953118023287825098194558153017567173613320698112509961818815930416903" +
51             "51598888519345807273866738589422879228499892086805825749279610484198444363463244" +
52             "96848756023362482704197862320900216099023530436994184914631409343173814364054625" +
53             "31520961836908887070167639642437814059271456354906130310720851038375051011574 77" +
54             "04171898610687396965521267154688957035035402123407849819334321068170121005627880" +
55             "23519303322474501585390473041995777709350366041699732972508868769664035557071622" +
56             "68447162560798826517871341951246652010305921236677194325278675398558944896970964" +
57             "09754591856956380236370162112047742728364896134225164450781824423529486363721 41" +
58             "74023889344124796357437026375529444833799801612549227850925778256209262264832627" +
59             "79333865664816277251640191059004916449982893150566047258027786318641551956532442" +
60             "58698294695930801915298721172556347546396447910145904090586298496791287406870504" +
61             "89585867174798546677575732056812884592054133405392200011378630094556068816674001" +
62             "69842055804033637953764520304024322566135278369511778838638744396625322498506549" +
63             "95886234281899707733276171783928034946501434558897071942586398772754710962953741" +
64             "52111513683506275260232648472870392076431005958411661205452970302364725492966693" +
65             "81151373227536450988903136020572481765851180630364428123149655070475102544650 11" +
66             "72721155519486685080036853228183152196003735625279449515828418829478761085263981 4";
67 }
```

Mathcore.java